MTI #1758
Docket No. 18153.0046

REGISTER POINTER TRAP

FIELD OF THE INVENTION:

The present invention relates to systems and methods for preventing processing errors in processors and, more particularly, to systems and methods for providing register pointer traps to identify registers having invalid data to prevent propagating processing errors due to these registers being used as pointers.

BACKGROUND OF THE INVENTION:

After power is applied to a chip, registers and memory elements in general on the chip will assume values that are unpredictable and will contain data that is invalid. When the chip is a processor, this is not problematic unless a program resident on the processor tries to read memory elements prior to known values having been written into the memory elements. This may be particularly problematic when programs use a memory element, such as a register, as a pointer to a memory. In the latter scenario, the pointer is invalid, and its use will lead to processing errors.

To prevent processing errors, there is a need for a mechanism to identify when a memory element, such as a register, has invalid data in it. There is a further need to flag registers that have invalid data in them to prevent these registers from being used as pointers to memory.

SUMMARY OF THE INVENTION:

According to embodiments of the present invention, trap flags and a pointer trap are associated with registers in a processor. Each trap flag indicates whether a corresponding register has been written with valid data. If not, the trap flag is set to indicate that the register

corresponding to the trap flag contains invalid data. During instruction processing, the pointer trap receives control signals from instruction fetch/decode logic on the processor indicating an instruction being processed calls for a register to be used as a pointer. If the specified pointer register has its corresponding trap flag set, the then the pointer trap indicates that a processing exception has occurred. The interrupt logic/exception processing logic then causes a trap interrupt service routine (ISR) to be executed in response to the exception. The ISR prevents errors from being propagated during the ensuing instruction processing due to invalid pointer values.

According to one embodiment of the invention, a method of preventing processing errors due to invalid pointers includes fetching an instruction having a pointer operand for execution. The method further includes determining whether a trap flag corresponding to the pointer is in a set or reset condition and generating a trap control signal when the trap flag is in a reset condition. The method may further include executing the instruction and triggering a trap interrupt based on the trap control signal.

The trap flag corresponding to the pointer may be changed from the reset condition to the set condition based upon a write to the pointer. In addition, the trap flag may be changed to the reset condition after a power up or a reset of the processor.

According to another embodiment of the invention, a method of preventing processing errors due to invalid pointers includes providing trap flags, each corresponding to a pointer register. The trap flags are reset upon a power up or reset. Conversely, the trap flag corresponding to each pointer register are set based on the pointer register being written. A trap control signal may be generated when an instruction reads a pointer register with a trap flag set to reset. The trap control signal may trigger a trap interrupt.

According to still another embodiment of the invention, a processor prevents processing errors due to invalid pointers. The processor includes instruction fetch and decode logic, registers, trap flags and pointer trap logic. The instruction fetch and decode logic fetches and decodes instructions. The trap flags each correspond to a register and each indicate a set or reset condition. The pointer trap is coupled to the trap flags and generates a trap control signal based on decoding an instruction that reads a register that has a corresponding trap flag in a reset condition. The trap control signal may only be generated when the register being read from is acting as a pointer register.

BRIEF DESCRIPTION OF THE FIGURES:

The above described features and advantages of the present invention will be more fully appreciated with reference to the detailed description and appended figures in which:

Fig. 1 depicts a functional block diagram of an embodiment of a processor chip within which embodiments of the present invention may find application.

Fig. 2 depicts a functional block diagram of a data busing scheme for use in a processor, which has a microcontroller and a digital signal processing engine, within which embodiments of the present invention may find application.

Fig. 3 depicts a functional block diagram of a processor configuration for implementing a pointer trap.

Fig. 4 depicts a method of processing instructions for detecting invalid pointers.

DETAILED DESCRIPTION:

According to embodiments of the present invention, trap flags and a pointer trap are associated with registers in a processor. Each trap flag indicates whether a corresponding register has been written with valid data. If not, the trap flag is set to indicate that the register corresponding to the trap flag contains invalid data. During instruction processing, the pointer trap receives control signals from instruction fetch/decode logic on the processor indicating an instruction being processed calls for a register to be used as a pointer. If the specified pointer register has its corresponding trap flag set, the then the pointer trap indicates that a processing exception has occurred. The interrupt logic/exception processing logic then causes a trap interrupt service routine (ISR) to be executed in response to the exception. The ISR prevents errors from being propagated during the ensuing instruction processing due invalid pointer values.

In order to describe embodiments of the invention, an overview of pertinent processor elements is first presented with reference to Figs. 1 and 2. The pointer trap functionality and interrupt processing is then described more particularly with reference to Figs. 3-4.

Overview of Processor Elements

Fig. 1 depicts a functional block diagram of an embodiment of a processor chip within which the present invention may find application. Referring to Fig. 1, a processor 100 is coupled to external devices/systems 140. The processor 100 may be any type of processor including, for example, a digital signal processor (DSP), a microprocessor, a microcontroller or combinations thereof. The external devices 140 may be any type of systems or devices including input/output devices such as keyboards, displays, speakers, microphones, memory, or other systems which

may or may not include processors. Moreover, the processor 100 and the external devices 140 may together comprise a stand alone system.

The processor 100 includes a program memory 105, an instruction fetch/decode unit 110, instruction execution units 115, data memory and registers 120, peripherals 125, data I/O 130, and a program counter and loop control unit 135. The bus 150, which may include one or more common buses, communicates data between the units as shown.

The program memory 105 stores software embodied in program instructions for execution by the processor 100. The program memory 105 may comprise any type of nonvolatile memory such as a read only memory (ROM), a programmable read only memory (PROM), an electrically programmable or an electrically programmable and erasable read only memory (EPROM or EEPROM) or flash memory. In addition, the program memory 105 may be supplemented with external nonvolatile memory 145 as shown to increase the complexity of software available to the processor 100. Alternatively, the program memory may be volatile memory which receives program instructions from, for example, an external non-volatile memory 145. When the program memory 105 is nonvolatile memory, the program memory may be programmed at the time of manufacturing the processor 100 or prior to or during implementation of the processor 100 within a system. In the latter scenario, the processor 100 may be programmed through a process called in-line serial programming.

The instruction fetch/decode unit 110 is coupled to the program memory 105, the instruction execution units 115 and the data memory 120. Coupled to the program memory 105 and the bus 150 is the program counter and loop control unit 135. The instruction fetch/decode unit 110 fetches the instructions from the program memory 105 specified by the address value contained in the program counter 135. The instruction fetch/decode unit 110 then decodes the

fetched instructions and sends the decoded instructions to the appropriate execution unit 115. The instruction fetch/decode unit 110 may also send operand information including addresses of data to the data memory 120 and to functional elements that access the registers.

The program counter and loop control unit 135 includes a program counter register (not shown) which stores an address of the next instruction to be fetched. During normal instruction processing, the program counter register may be incremented to cause sequential instructions to be fetched. Alternatively, the program counter value may be altered by loading a new value into it via the bus 150. The new value may be derived based on decoding and executing a flow control instruction such as, for example, a branch instruction. In addition, the loop control portion of the program counter and loop control unit 135 may be used to provide repeat instruction processing and repeat loop control as further described below.

The instruction execution units 115 receive the decoded instructions from the instruction fetch/decode unit 110 and thereafter execute the decoded instructions. As part of this process, the execution units may retrieve one or two operands via the bus 150 and store the result into a register or memory location within the data memory 120. The execution units may include an arithmetic logic unit (ALU) such as those typically found in a microcontroller. The execution units may also include a digital signal processing engine, a floating point processor, an integer processor or any other convenient execution unit. A preferred embodiment of the execution units and their interaction with the bus 150, which may include one or more buses, is presented in more detail below with reference to Fig. 2.

The data memory and registers 120 are volatile memory and are used to store data used and generated by the execution units. The data memory 120 and program memory 105 are preferably separate memories for storing data and program instructions respectively. This format

is a known generally as a Harvard architecture. It is noted, however, that according to the present invention, the architecture may be a Von-Neuman architecture or a modified Harvard architecture which permits the use of some program space for data space. A dotted line is shown, for example, connecting the program memory 105 to the bus 150. This path may include logic for aligning data reads from program space such as, for example, during table reads from program space to data memory 120.

Referring again to Fig. 1, a plurality of peripherals 125 on the processor may be coupled to the bus 125. The peripherals may include, for example, analog to digital converters, timers, bus interfaces and protocols such as, for example, the controller area network (CAN) protocol or the Universal Serial Bus (USB) protocol and other peripherals. The peripherals exchange data over the bus 150 with the other units.

The data I/O unit 130 may include transceivers and other logic for interfacing with the external devices/systems 140. The data I/O unit 130 may further include functionality to permit in circuit serial programming of the Program memory through the data I/O unit 130.

Fig. 2 depicts a functional block diagram of a data busing scheme for use in a processor 100, such as that shown in Fig. 1, which has an integrated microcontroller arithmetic logic unit (ALU) 270 and a digital signal processing (DSP) engine 230. This configuration may be used to integrate DSP functionality to an existing microcontroller core. Referring to Fig. 2, the data memory 120 of Fig. 1 is implemented as two separate memories: an X-memory 210 and a Y-memory 220, each being respectively addressable by an X-address generator 250 and a Y-address generator 260. The X-address generator may also permit addressing the Y-memory space thus making the data space appear like a single contiguous memory space when addressed

from the X address generator. The bus 150 may be implemented as two buses, one for each of the X and Y memory, to permit simultaneous fetching of data from the X and Y memories.

The W registers 240 are general purpose address and/or data registers. The DSP engine 230 is coupled to both the X and Y memory buses and to the W registers 240. The DSP engine 230 may simultaneously fetch data from each the X and Y memory, execute instructions which operate on the simultaneously fetched data and write the result to an accumulator (not shown) and write a prior result to X or Y memory or to the W registers 240 within a single processor cycle.

In one embodiment, the ALU 270 may be coupled only to the X memory bus and may only fetch data from the X bus. However, the X and Y memories 210 and 220 may be addressed as a single memory space by the X address generator in order to make the data memory segregation transparent to the ALU 270. The memory locations within the X and Y memories may be addressed by values stored in the W registers 240.

Any processor clocking scheme may be implemented for fetching and executing instructions. A specific example follows, however, to illustrate an embodiment of the present invention. Each instruction cycle is comprised of four Q clock cycles Q1 – Q4. The four phase Q cycles provide timing signals to coordinate the decode, read, process data and write data portions of each instruction cycle.

According to one embodiment of the processor 100, the processor 100 concurrently performs two operations – it fetches the next instruction and executes the present instruction. Accordingly, the two processes occur simultaneously. The following sequence of events may comprise, for example, the fetch instruction cycle:

Q1: Fetch Instruction
Q2: Fetch Instruction

Q3: Fetch Instruction
Q4: Latch Instruction into prefetch register, Increment PC

The following sequence of events may comprise, for example, the execute instruction cycle for a single operand instruction:

Q1: latch instruction into IR, decode and determine addresses of operand data
Q2: fetch operand
Q3: execute function specified by instruction and calculate destination address for data
Q4: write result to destination

The following sequence of events may comprise, for example, the execute instruction cycle for a dual operand instruction using a data pre-fetch mechanism. These instructions pre-fetch the dual operands simultaneously from the X and Y data memories and store them into registers specified in the instruction. They simultaneously allow instruction execution on the operands fetched during the previous cycle.

Q1: latch instruction into IR, decode and determine addresses of operand data
Q2: pre-fetch operands into specified registers, execute operation in instruction
Q3: execute operation in instruction, calculate destination address for data
Q4: complete execution, write result to destination

Pointer Trap Embodiments

Fig. 3 depicts a functional block diagram of a processor configuration for implementing a pointer trap. Referring to Fig. 3, the processor includes a program memory 300 with instructions 301 and 302 in a processing sequence. A program counter 305 is coupled to the program memory 300 and indicates the current instruction. An instruction fetch/decoder unit 310 is coupled to the program memory 300 and fetches the next instruction for execution as indicated by the program counter 305.

A reset/power up control unit 315 resides on the processor and is used to initialize the processor after a power up and after a processor reset. The unit 315 may be used to control when the processor begins to process instructions after a power on event. This may be performed by letting a predetermined number of clock cycles occur prior to releasing the processor to process instructions. During this start up period, the processor and oscillator clocking the processor are given time to stabilize. In the event of either a power on or a processor reset, and possibly other events, the reset/power up control unit 315 sends a reset control signal to the trap flag control unit 320 to reset the trap flags. This is done because the registers on the processor contain invalid data after a power on event or after a processor reset.

The trap flag control unit 320 is coupled to the reset/power up control unit 315 and the instruction fetch/decode unit 310. In response to receiving a reset control signal from the reset/power up control unit 315, the trap flag control unit 320 causes the trap flags to be reset to a predetermined value, which may be for example a zero or a one. The reset causes the trap flags 345 to indicate that their corresponding registers 340 include invalid data and hence should not be used to provide pointer values.

When an instruction that is being processed calls for writing to one of the registers, the instruction fetch/decode unit 310 generates a write control signal that identifies the particular registers that are being written into. The trap flag control unit 320 receives the write control signal and sets the trap flag 345 for the corresponding register that is being written into to a value that indicates that the register contains valid data. This value may be a zero or one and is the opposite value from the reset value stored after a processor power up or reset.

The trap flags 345 accordingly each relate to a corresponding register 340. In a reset state, a trap flag indicates that the corresponding register contains invalid data. In a set state, the

trap flag indicates that the corresponding register contains valid data. The state of the trap flags 345 are reset upon a power on occurrence or a processor reset. The state of each trap flag 345 is thereafter changed to a set state after the corresponding register is written.

The pointer trap 325 is coupled to the instruction fetch/decode unit 310. When an instruction that is being processed calls for reading a pointer value (or other value) from one of the registers, the instruction fetch/decode unit 310 generates a read control signal that identifies the particular registers that are being written into. The pointer trap 325 receives the read control signal for the corresponding register that is being read from. The pointer trap 325 reads the trap flag of the register being read that is identified in the read control signal. When the trap flag corresponding to the register being read is in the reset condition, the pointer trap generates a trap control signal that is sent to the interrupt logic 330 to cause a trap interrupt. When the trap flag corresponding to the register being read is in the set condition, the pointer trap generates a trap control signal that is sent to the interrupt logic 330 that does not cause a trap interrupt. In this manner, the pointer trap 325 monitors instructions that are being processed and when the instructions will result in reading a pointer value that includes invalid data, a trap interrupt is triggered.

The interrupt logic 330 receives indications of exceptions from various units on the processor. The interrupt logic 330 thereafter arbitrates priority and determines which interrupt to service and when. Upon selection of the trap interrupt, the interrupt logic 330 identifies the trap interrupt to the ISR vector register 335.

The ISR vector register 335 is a look up table that includes the address of the first instruction of various interrupt service routines (ISRs). When the trap interrupt is being serviced, the address of the first instruction of the trap ISR is loaded into the program counter 305 which

causes the trap ISR to be run. The trap ISR typically places the processor in a known condition to avoid processing errors.

Fig. 4 depicts a method of processing instructions for detecting invalid pointers. Referring to Fig. 4, in step 400, the processor fetches an instruction. In step 410, the processor decodes the instruction. Then in step 420, the processor identifies pointers in the instruction operands. This may be performed by an instruction decoder on the processor. In step 430, the instruction decoder generates read control signals identifying pointer registers that are being read from by a current instruction. The read control signals generated may be sent to a pointer trap unit. In step 440, the processor checks the trap flag value corresponding to one or more of the pointer registers that are being read from. This may be performed by a pointer trap unit. In step 450, the processor determines whether the trap flag is set for one or more of the corresponding pointer registers being read from. If so, then step 460 begins. If not, then step 480 begins and the processor executes the instruction.

In step 460, the processor sends a trap control signal to interrupt control logic on the processor. The trap control signal indicates that a pointer trap exception has occurred. Then in step 470, the processor loads and executes a trap ISR to prevent propagating processing errors and to return the processor to a known state.

While specific embodiments of the present invention have been illustrated and described, it will be understood by those having ordinary skill in the art that changes may be made to those embodiments without departing from the spirit and scope of the invention. For example, Fig. 4 depicts an embodiment of the present invention where the instruction is prevented from being executed when the pointer trap flag is set. In alternative embodiments of the invention, the instruction may be executed even if the pointer trap flag is set. In this embodiment, the trap ISR

may cause the processor to undo or correct for the instruction having been executed. In still other embodiments, an instruction may be executed when a trap flag is set but, in the case of a write instruction, the write to memory may be inhibited by a trap control signal. Many other variations are possible and within the scope of the present invention.

5